

Universidad Nacional de Mar del Plata
Facultad de Ingeniería

PROYECTO FINAL

"SUSÍ QUÉ ME LLAMAS"



GRUPO n° 12 - SOFÍA ISABELLA PALLADINO

Programación III

Ciclo 2024

Índice

Introducción.....	1
Desarrollo.....	1
Metodología.....	1
Diseño e Implementación.....	2
Diagrama UML.....	2
Diseño MVC.....	2
Patrones Utilizados.....	3
Diseño de clases.....	9
Conclusión.....	17



Introducción

Al alumno se le presenta la consigna de desarrollar un sistema informático para una empresa de traslados de pasajeros, siendo la interfaz de usuario la computadora. El sistema pedido debe gestionar tanto la parte de la información de una empresa de transporte de pasajeros como la de transporte de mensajería.

El sistema es desarrollado mediante el lenguaje de programación orientada a objetos (POO) Java®.

El objetivo académico de este proyecto, es profundizar en los temas dados por la cátedra a lo largo del presente cuatrimestre implementando conceptos como jerarquía de clases, herencia, polimorfismo y la reutilización. El manejo de estos conceptos de forma práctica, permite ayudar a comprender de manera más clara y práctica cómo funcionan estos conceptos en la programación orientada a objetos.

Desarrollo

Metodología

Para la primera parte del enunciado pedido por la cátedra, aparecieron desacuerdos con los integrantes del grupo al que pertenecía anteriormente debido a la falta de comunicación no se concertaron las actividades de cada integrante, generando que el trabajo no se divida de forma equitativa tanto en la logística como en la programación. El resultado de esto fue realizar una primera entrega con el 80% de código de mí autoría, del cual el resto de los integrantes no entendían.

El enfoque inicial de esta segunda parte fue persistiendo datos mediante la utilización de serialización (usando XML) y aplicando los patrones DTO y DAO. Luego, en la implementación del patrón MVC para comenzar con la lógica de las ventanas y los controladores; aplicando en general los métodos de las clases que en un principio eran de “Gestión” (GestionPedido, GestionChoferes, GestionViajes) utilizando en los controladores.

Mediante el desarrollo de Controladores, se desarrollan las Vistas, por lo que al avanzar en la lógica, también se avanza en la parte visual, por ejemplo, una vez que se termina con toda la lógica para el registro del usuario, se crea la ventana para que permita que cualquiera que ejecute el programa pueda crear un usuario. Acorde avanzaba en esta parte del código es cuando comencé a aplicar el patrón Observer-Observable y Threads.

Por último se tuvo en cuenta la corrección de la primera parte y se actualizó el código para cumplir con lo pedido, ademas al recorrer todo pude eliminar las clases y/o metodos innecesarios..



Diseño e Implementación

Diagrama UML

Adjunto UML en foto, en el repositorio de GitHub

Diseño MVC

El modelo de 3 capas a MVC se adaptó teniendo como base el identificar que partes de la lógica de la primera parte del TP pasaría a formar parte de los modelos, vistas y controladores.

Muchos métodos que se encontraban en Gestión...(viajes,pedidos, choferes) fueron utilizados en los controladores encargados de dichas gestiones. En paralelo con los controladores se fueron creando las ventanas (o vistas).

Mientras implementaba MVC se presentaron problemas en donde era necesario el uso de hilos y del patron observer/observable.

Al implementar MVC, las clases Gestión.. quedaron obsoletas.



Patrones Utilizados

FACTORY

El patrón Factory fue utilizado para crear instancias de Usuarios, Viajes y Vehículos. Esto es importante porque se debe considerar diferentes tipos de usuarios (administrador, chofer o cliente), diferentes tipos de viajes (zona estándar, peligrosa, sin asfaltar) y diferentes tipos de vehículos (Combi, Moto y Automóvil). Esto permite centralizar la creación en una fábrica especializada según los parámetros dados, lo que mejora la claridad y permite la extensión futura del sistema.



La clase usuarioFactory recibe un Usuario, si el atributo de Usuario “isAdmin()” es true (es un atributo booleano) entonces crea un administrador, sino crea un cliente.

```
package org.usuario;
public class UsuarioFactory { ↗ SofialPalladino
    public Usuario crea(Usuario usuario) { 2 usages ↗ SofialPalladino
        if (usuario.isAdmin())
            return new Administrador(usuario);
        else
            return new Cliente(usuario);
    }
}
```

La clase ViajeFactory implementa el metodo getViaje que recibe un pedido; dependiendo lo que se solicita en el pedido se crea un viaje que cumpla con todos los requisitos del pedido.



```
public class ViajeFactory { * SofialPalladino

    public IViaje getViaje(Pedido pedido) { * 1 usage * SofialPalladino
        IViaje viaje;
        viaje=null;
        if(pedido.getZona().equalsIgnoreCase( anotherString: "Zona Estandar")) {
            viaje=new ViajeZonaEstandar(pedido);
        }
        else if(pedido.getZona().equalsIgnoreCase( anotherString: "Calle sin asfaltar")) {
            viaje=new ViajeZonaCalleSinAfaltar(pedido);
        }
        else if(pedido.getZona().equalsIgnoreCase( anotherString: "Zona Peligrosa")){
            viaje=new ViajeZonaPeligrosa(pedido);
        }

        if(pedido.getMascota()) {
            viaje=new DecoratorConMascota(viaje);
        }

        if(pedido.getEquipaje().equalsIgnoreCase( anotherString: "Baul")) {
            viaje=new DecoratorEquipajeBaul(viaje);
        }

        return viaje;
    }
}
```

La clase vehiculoFactory implementa el método getVehiculo que recibe por parámetro el tipo de automovil del que se creara el objeto automovil y su patente, implementando el factory para la creacion de dicho objeto

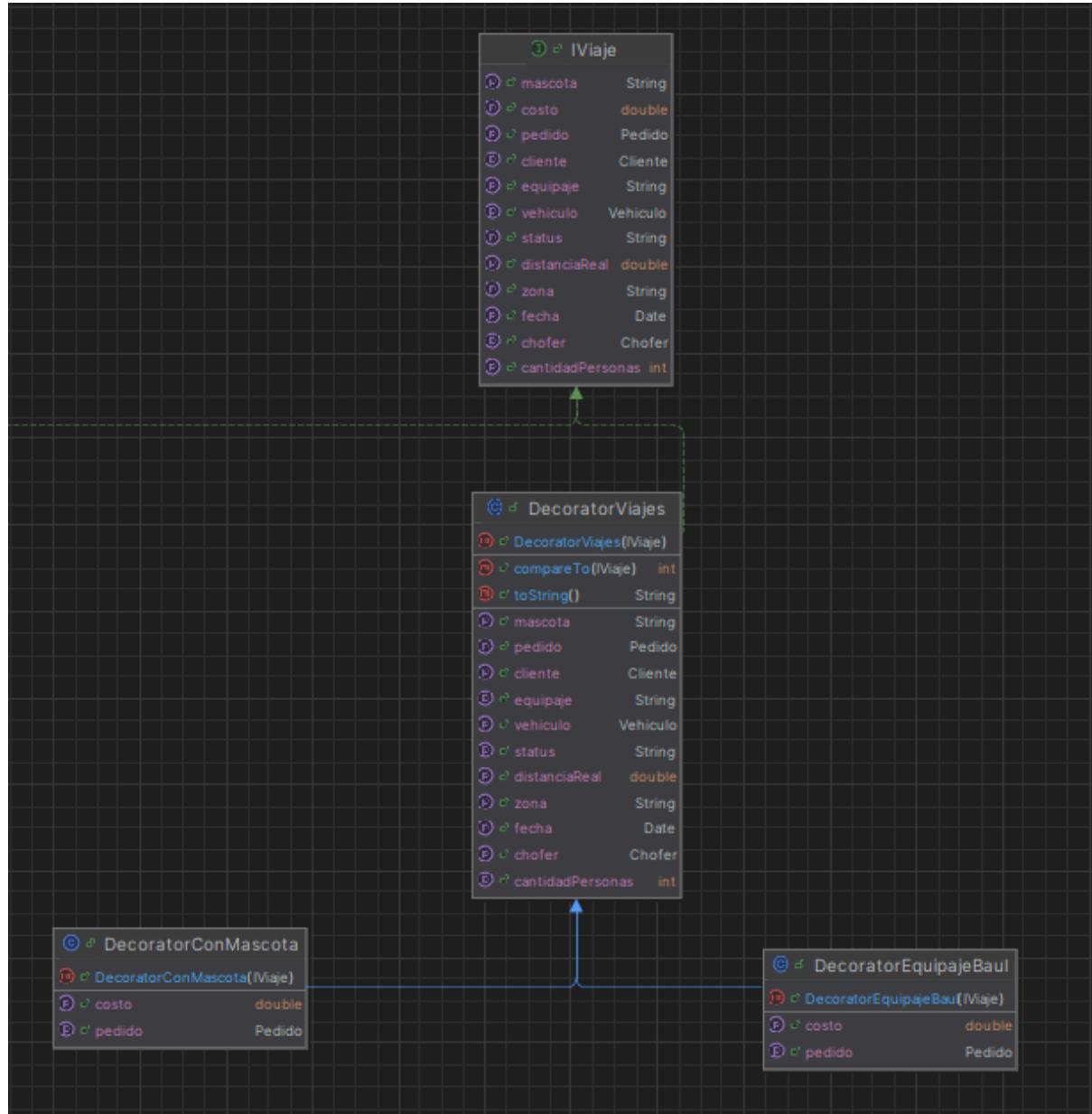
```
public class VehiculoFactory { * SofialPalladino

    public Vehiculo getVehiculo(String tipo, String patente) {
        if (Objects.equals(tipo, "Automovil"))
            return new Automovil(patente);
        if (Objects.equals(tipo, "Moto"))
            return new Moto(patente);
        if (Objects.equals(tipo, "Combi"))
            return new Combi(patente);
        return null;
    }
}
```



DECORATOR

El patrón Decorator desempeña un papel fundamental en la creación y personalización de los Viajes dentro del sistema. Su función principal radica en la capacidad de agregar dinámicamente características y funcionalidades adicionales a los Viajes (viéndose en el código anterior), según lo requerido en el Pedido. Esta capacidad de añadir características nuevas de forma dinámica es especialmente valiosa para evitar la conocida explosión clases, ya que permite construir Viajes únicos y personalizados sin la necesidad de crear una gran cantidad de clases diferentes.





```
package org.viaje;

import org.pedido.Pedido;

public class DecoratorConMascota extends DecoratorViajes { ▲ SofialPalladino *

    public DecoratorConMascota(IViaje encapsulado) { 1 usage ▲ SofialPalladino
        super(encapsulado);
    }

    public double getCosto() { 8 usages ▲ SofialPalladino

        double costoEncapsulado=this.encapsulado.getCosto();
        double incrXPersona=Viaje.getCostoBase()*0.10*this.encapsulado.getPedido().getCantPersonas();
        double incrXKm=Viaje.getCostoBase()*0.20*this.encapsulado.getDistanciaReal();

        return costoEncapsulado+incrXPersona+incrXKm;
    }

    public Pedido getPedido() { ▲ SofialPalladino
        return this.encapsulado.getPedido();
    }
}
```

```
package org.viaje;

import org.pedido.Pedido;

public class DecoratorEquipajeBaul extends DecoratorViajes { ▲ SofialPalladino *

    public DecoratorEquipajeBaul(IViaje encapsulado) { 1 usage ▲ SofialPalladino
        super(encapsulado);
    }

    public double getCosto() { 8 usages ▲ SofialPalladino
        double costoEncapsulado=this.encapsulado.getCosto();
        double incrXPersona=Viaje.getCostoBase()*0.10*this.encapsulado.getPedido().getCantPersonas();
        double incrXKm=Viaje.getCostoBase()*0.05*this.encapsulado.getDistanciaReal();

        return costoEncapsulado+incrXPersona+incrXKm;
    }

    public Pedido getPedido() { ▲ SofialPalladino
        return this.encapsulado.getPedido();
    }
}
```



```
package org.viaje;

import org.chofer.Chofer;
import org.pedido.Pedido;
import org.usuario.Cliente;
import org.vehiculo.Vehiculo;
import java.util.Date;

public abstract class DecoratorViajes implements IViaje { 2 inheritors  ↗ SofialPalladino *

    protected IViaje encapsulado;  21 usages

    public DecoratorViajes(IViaje encapsulado) { 2 usages  ↗ SofialPalladino
        this.encapsulado=encapsulado;
    }

    public double getDistanciaReal() { 6 usages  ↗ SofialPalladino
        return this.encapsulado.getDistanciaReal();
    }

    public Pedido getPedido() { 2 overrides  ↗ SofialPalladino *
        return this.encapsulado.getPedido();
    }

    public void setVehiculo(Vehiculo vehiculo) { new *
        this.encapsulado.setVehiculo(vehiculo);
    }
}
```

SINGLETON

El patrón Singleton se utilizó para la creación de la clase Empresa, de esta forma se asegura que exista una única instancia de esta a la vez que proporcionamos acceso global a ella. Es fundamental para que haya una gestión eficiente de choferes, vehículos, clientes y otros recursos de la empresa de transporte.

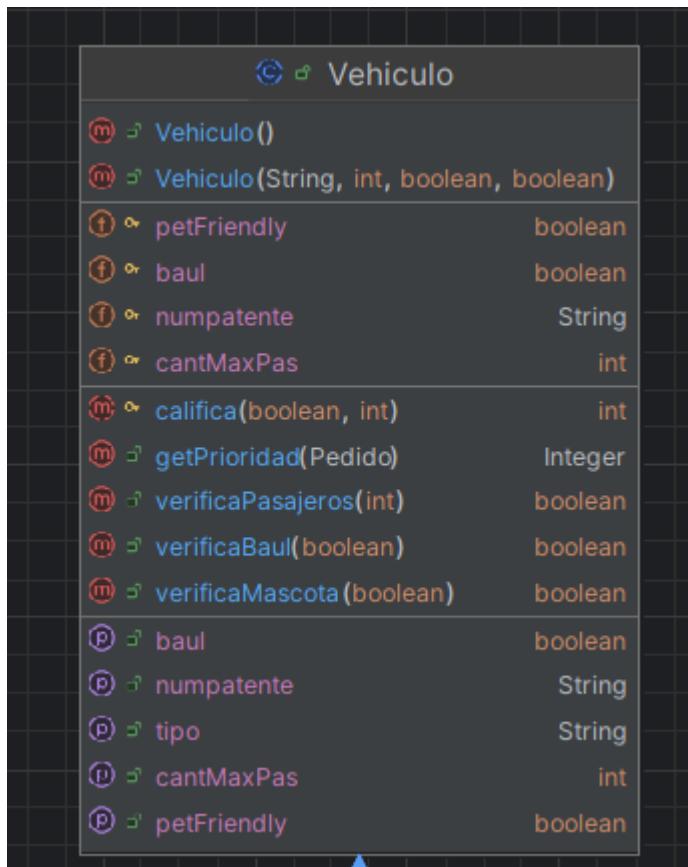
```
public static synchronized Empresa getInstance() {  ↗ SofialPalladino *
    if (instance == null)
        instance = new Empresa();
    return instance;
}
```

TEMPLATE

El patron Template fue util para establecer la prioridad a la hora de buscar un vehículo que sea apto para el pedido. El método getPrioridad de la clase Vehiculo determina la prioridad de un objeto Pedido en función de ciertos criterios (la cantidad de personas, si utiliza o no el baul, si lleva o no



mascota). Si el pedido cumple con todos los criterios, calcula y devuelve la prioridad; de lo contrario, devuelve null.



```
public Integer getPrioridad(Pedido pedido) {
    Integer valorPrioridad = null;
    if(verificaPasajeros(pedido.getCantPersonas()) && verificaBaul(pedido.usoBaul()) && verificaMascota(pedido.getMascota())) {
        valorPrioridad = Integer.valueOf(califica(pedido.usoBaul(), pedido.getCantPersonas()));
    }
    return valorPrioridad;
}
```

MVC

El patrón MVC se utilizó para poder separar la aplicación en tres partes, el modelo donde se encuentra toda la lógica del programa, la vista que es la responsable de interactuar con el usuario tanto mostrando información como método de entrada al sistema y por último el controlador que se encarga de actuar como intermediario entre estas últimas dos capas manejando que vistas y datos se muestran

OBSERVER/OBSERVABLE

El patrón Observer/Observable se utilizó como complemento al MVC, ya se encargó de notificar automáticamente los cambios presentes en el Modelo a las Vistas

DTO



El patrón DTO se utilizó para poder separar claramente las capas de la aplicación, especialmente entre el modelo y la capa de persistencia. Esta separación permite una mayor modularidad y aumenta el control sobre los datos que se almacenan

DAO

El patrón DAO fue utilizado para poder persistir y leer de archivo los datos de EmpresaDTO, el cual contiene todos los datos necesarios para correr la aplicación

Diseño de clases

Para diseñar las distintas clases de mi aplicación decidí guiarme por el principio SOLID, de esta manera creaba clases lo más independientes separadas entre sí para disminuir el acoplamiento entre ellas. De esta manera decidí separar en módulos independientes las distintas tareas que tiene que hacer el Administrador y los Usuarios del sistema, de esta manera si en un futuro quiero agregar mas tareas poder hacerlo de una manera que no rompa con la lógica actual de la ap.
Lo hice creando las clases (GestionPedido, GestionPagoChoferes, GestionPuntajes, GestionUsuarios, GestionViajes).

Paquetes y clases creadas

CHOFER

```
✓  chofer
    ⓒ Chofer
    ⓒ ChoferContratado
    ⓒ ChoferPermanente
    ⓒ ChoferTemporario
    ⓒ CrearChoferHilo
    ⓒ GestionPagoChoferes
```

En el paquete choferes cree la clase Chofer, la cual es una clase abstracta que contiene todos los atributos requeridos por los choferes en mi aplicación. De esta clase heredan todas las clases Chofer.

También creé la clase **GestionPagoChoferes**, que es el encargado de calcular el sueldo a ser depositado en la cuenta de cada chofer según el tipo del mismo, los viajes realizados y lo recaudado

Clase Chofer:

Este método busca un viaje sin chofer en la empresa, asigna el chofer actual a ese viaje, y actualiza los estados y atributos relacionados del viaje y del chofer.



Contrato: public synchronized void buscaViajeChofer() throws NoChoferException.

Clase Chofer:

Contrato: public synchronized void buscaViajeChofer() throws NoChoferException

Precondición:

- **Instancia de Empresa:** Debe existir una instancia de Empresa ya inicializada.
- **Lista de Viajes Sin Chofer:** La lista de viajes sin chofer debe estar correctamente inicializada
- **Estado del Chofer:** El objeto **this** debe ser una instancia de una clase que implementa el comportamiento de un chofer (por ejemplo, **ChoferContratado**).

Postcondiciones

- **Asignación de Chofer:** Si hay viajes sin chofer, el primer viaje de la lista se asignará al chofer actual (**this**), y su estado se actualizará a "Iniciado".
- **Actualización de Estado del Chofer:** El estado del chofer se actualizará a "En Viaje".
- **Recaudación y Actualización de Atributos:** Si el chofer es una instancia de **ChoferContratado**, se actualizará la recaudación del viaje. Además, se incrementarán los viajes y los kilómetros del chofer.

Excepciones

- **NoChoferException:** Esta excepción se lanza si no se puede asignar un chofer a un viaje

```
public class GestionPagoChoferes { ▲ SofialPalladino *  
  
    Empresa e=Empresa.getInstance(); 1 usage  
  
    public void calculoPagoChoferes(Usuario usuario) { 1 usage ▲ SofialPalladino *  
        if (usuario.getClass().equals(Administrador.class)) {  
            double sueldo=0;  
            double totalsueldos=0;  
            Chofer choferaux=null;  
  
            List<Chofer> choferes=e.getChoferes();  
            for(int i=0;i<choferes.size();i++) {  
                choferaux=choferes.get(i);  
                sueldo=choferaux.getSueldo();  
                totalsueldos+=sueldo;  
                System.out.println("El chofer: "+choferaux.getNombre()+" tiene que cobrar $" + sueldo);  
            }  
            System.out.println("Total de dinero a pagar por los sueldos: "+totalsueldos);  
        }else {  
            System.out.println("Solo el administrador puede pagar los salarios");  
        }  
    }  
}
```

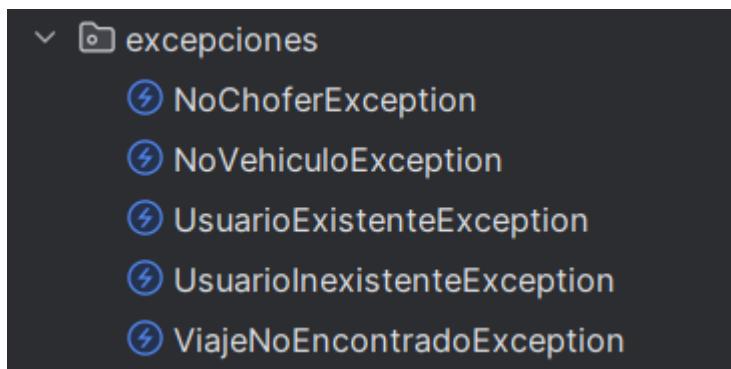


Mediante el método “calculoPagoChoferes” la clase se encarga de calcular el sueldo de cada chofer, pudiendo calcularlo siempre y cuando el usuario que quiera implementar el método sea un administrador (quien es el responsable del pago de choferes)

Contrato: public void calculoPagoChoferes(Usuario usuario)

Precondición:

- **Instancia de Empresa:** Debe existir una instancia de Empresa ya inicializada.
- **Usuario Valido:** El parámetro usuario debe ser una instancia de la clase Administrador.
- **Postcondiciones:**
- **Usuario Administrador:** Si el usuario es un administrador, se calcula el pago de cada chofer y se muestra en la consola.
- **Pago total:** Se imprime el total de dinero a pagar por los sueldos de los choferes.
- **Usuario Cliente:** Si el usuario no es un administrador, se muestra un mensaje indicando que solo el administrador puede realizar esta acción.



En el paquete excepciones crea todas las posibles excepciones que podrá lanzar mi aplicación (eso pasa las veces que: no exista un chofer a asignar, no exista un vehículo a asignar, el usuario a registrar ya exista, el usuario a ingresar no exista, no se encuentre un viaje)

ejemplo de una de las clases del paquete de excepciones

```
package org.excepciones;

public class NoVehiculoException extends Exception {    ▲ SofialPalladino
    public NoVehiculoException() {    ▲ SofialPalladino
        super("No se encontró ningún vehículo que cumpla con los requisitos del viaje.");
    }

    public NoVehiculoException(String message) {    ▲ SofialPalladino
        super(message);
    }
}
```

PAQUETE PEDIDO



```
▼ └ pedido
    └ GestionPedidos
        └ Pedido
```

En el paquete “pedido” cree únicamente dos clases, la clase “Pedido” que contiene toda la información del pedido del cliente y la clase “GestionPedidos” que tiene como método “buscaVehiculo” que se encarga de encontrar los vehículos que posee la empresa que cumplan con lo solicitado en el pedido del cliente.

```
public ArrayList<Vehiculo> buscarVehiculo(List<Vehiculo> v, int cantPasajeros, boolean baul, boolean mascota ) {
    ArrayList<Vehiculo> vehiculosCumplen = new ArrayList<Vehiculo>();
    int i = 0;
    Vehiculo p=null;
    while (i < v.size()) {
        p= v.get(i);
        if (p.verificaPasajeros(cantPasajeros) && p.verificaBaul(baul)&& p.verificaMascota(mascota)) {
            vehiculosCumplen.add(p);
        }
        i++;
    }
    return vehiculosCumplen;
}
```

Contrato: public ArrayList<Vehiculo> buscarVehiculo(List<Vehiculo> v, int cantPasajeros, boolean baul, boolean mascota) throws NoVehiculoException

Precondiciones:

- La lista de vehículos no debe ser nula.
- Los objetos en la lista de vehículos deben ser instancias válidas de la clase Vehiculo.
- cantPasajeros debe ser un número positivo.

Postcondiciones:

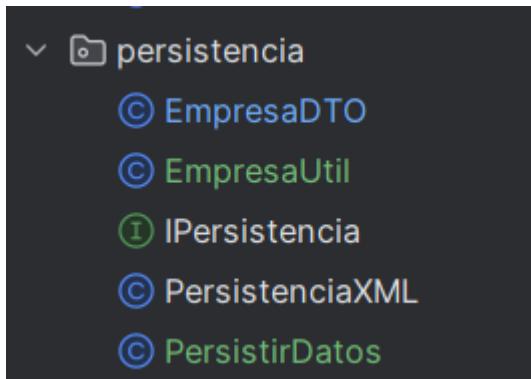
- Se crea una lista vehiculosCumplen que contiene todos los vehículos que cumplen con los criterios de capacidad de pasajeros, disponibilidad de baúl y capacidad de transporte de mascotas especificados.
- Se retorna la lista vehiculosCumplen, que puede estar vacía si no se encuentra ningún vehículo que cumpla con los criterios.

Excepciones

- **NoVehiculoException:** Se lanza si la lista de vehículos (v) es nula o vacía.

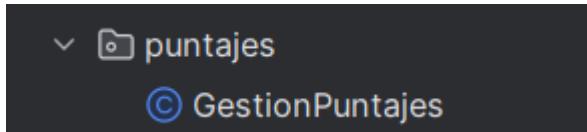


PAQUETE PERSISTENCIA

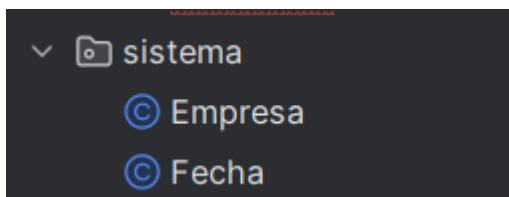


En el paquete persistencia cree todas las clases que me ayudan a almacenar y leer los datos de mi app.

Dentro de dicho paquete cree dos clases que me ayudan con el manejo de los datos, uno es “EmpresaUtil” que se encarga de crear el DTO de empresa y de llenar los datos de “EmpresaDTO” en mi clase Empresa y la clase “PersistirDatos” que tiene métodos para guardar y cargar los datos del sistema

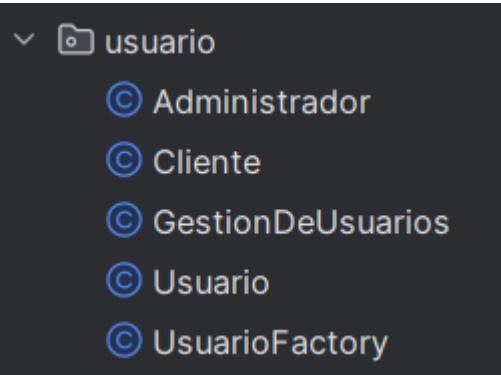


En el paquete puntajes se encuentra únicamente la clase “GestionPuntajes”, clase que es utilizada por el administrador para que se calculen los puntajes de todos los Choferes.



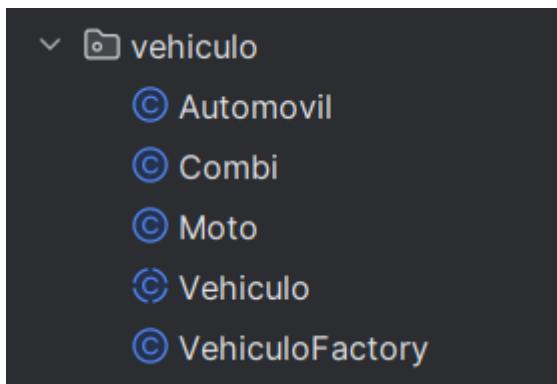
En el paquete sistema se encuentra la clase “Empresa”, encargada de tener toda la información de mi aplicación como también contar con la lógica necesaria para acceder a ella en todo el sistema como también a sus datos.

También se encuentra la clase abstracta “Fecha”, encargada de facilitar la creación de “Fechas” en todo el sistema

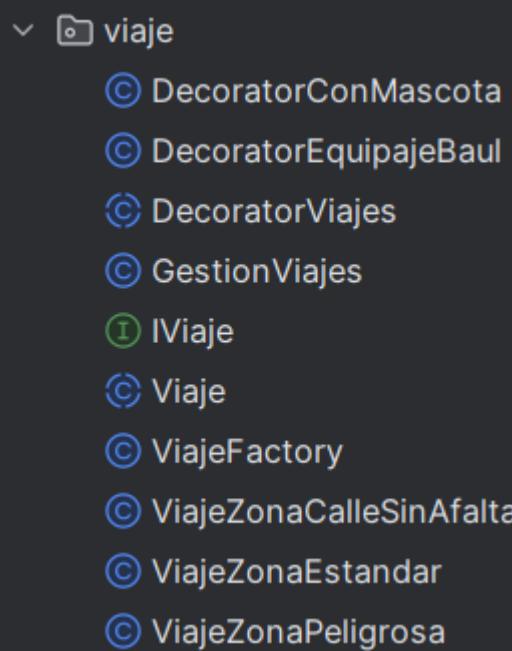


En el paquete `usuario` contamos con la clase abstracta `Usuario` que contiene toda la información del usuario base, la clase `Cliente` (que hereda de `Usuario`) y la clase `Administrador` (que hereda de `Usuario`), cada una de estas clases representan a los dos únicos tipos de usuarios presentes en el sistema.

También tenemos la clase “`UsuarioFactory`” que se encarga de crear a los Usuarios y la clase `GestionDeUsuarios` que nos permite cambiar los datos de los Usuarios



En el paquete `vehiculo` se tiene la clase “`VehiculoFactory`” que se encarga de la creación de los distintos Vehículos del sistema, tambien se encuentra la clase abstracta “`Vehiculo`” que contiene todos los datos para que las distintas clases la hereden.



En el paquete viaje se tiene la clase Viaje que hereda de IViaje que tiene los datos básicos para todos los viajes y contamos con todos los decoradores que le podemos aplicar a nuestra clase Viaje, cada decorador modifica la manera en la que se calcula el valor del viaje.

También se encuentra la clase GestionViajes, encargada de todas las operaciones comunes



Conclusión

A lo largo del desarrollo de este trabajo las expectativas que tenía en un principio fueron mutando a lo largo de la realización de este trabajo. Académicamente me hubiera gustado haber estado presente en las clases teóricas/prácticas aunque por motivos de salud no pude, igualmente agradezco los videos proporcionados por la cátedra que complementé con investigación en internet haciendo más amena partes del trabajo final..

En el aspecto de trabajo colaborativo claramente no cumplí las expectativas (tanto expectativas propias como para con los otros integrantes del grupo) pero igualmente la experiencia me servirá para encarar futuros proyectos/trabajos colaborativos de manera diferente.

Solución Destacada:

Me gustaría destacar toda la parte que se encarga de la “Gestión del Pedido”, todo el camino que se realiza desde que el cliente ingresa su pedido hasta que se crea (o no) un viaje que cumpla con dichas características para luego asignarle su vehículo más óptimo y un chofer. Esa lógica era la que más me preocupaba tener bien realizada.

Además de esto; cosas que en un principio pensé que serían sencillas como la distribución de cómo están hechas las ventanas también me supuso gran parte del tiempo.

Al no poder asistir a las consultas, lo solucioné investigando sobre el desarrollo de la parte gráfica de un programa, mirando videos en internet informandome como se hacía generalmente (y mediante mucha prueba y error) y después pudiendo adaptar lo aprendido a lo solicitado a hacer, también cumpliendo con la proyección que tenía de cómo debían ser las ventanas para que muestren la información de manera armoniosa y clara para el usuario.

Con el conocimiento adquirido hasta ahora plantearía nuevamente la organización de las clases y los métodos utilizados a lo largo del TP, ya que a medida que avanzaba con la elaboración de esta segunda parte tuve que retrasar el desarrollo arreglando problemas del diseño de la primera parte. Si bien el código entregado tiene sus fallos (los cuales por falta de tiempo no puedo perfeccionar) igualmente espero que se haya visto reflejado el trabajo que hubo detrás de la realización de este trabajo.